Understanding slice notation

Asked 11 years, 3 months ago Active 2 months ago Viewed 1.7m times

3254

1557

I need a good explanation (references are a plus) on Python's slice notation.

To me, this notation needs a bit of picking up.

It looks extremely powerful, but I haven't quite got my head around it.

python list slice iterable

edited May 8 '19 at 16:03
kmario23
31k ● 9 ● 101 ● 109

asked Feb 3 '09 at 22:31
Simon
63.8k ● 24 ● 80 ● 116

32 Answers

Active Oldest Votes

1 2 Next

4458

It's pretty simple really:

```
a[start:stop]  # items start through stop-1
a[start:]      # items start through the rest of the array
a[:stop]       # items from the beginning through stop-1
a[:]           # a copy of the whole array
```

There is also the `step` value, which can be used with any of the above:

```
a[start:stop:step] # start through not past stop, by step
```

The key point to remember is that the `:stop` value represents the first value that is *not* in the selected slice. So, the difference between `stop` and `start` is the number of elements selected (if `step` is 1, the default).

The other feature is that `start` or `stop` may be a *negative* number, which means it counts from the end of the array instead of the beginning. So:

```
a[-1]    # last item in the array
a[-2:]   # last two items in the array
a[:-2]   # everything except the last two items
```

Similarly, `step` may be a negative number:

```
a[::-1]    # all items in the array, reversed
a[1::-1]   # the first two items, reversed
```

```
a[:-3:-1]  # the last two items, reversed
a[-3::-1]  # everything except the last two items, reversed
```

Python is kind to the programmer if there are fewer items than you ask for. For example, if you ask for `a[:-2]` and `a` only contains one element, you get an empty list instead of an error. Sometimes you would prefer the error, so you have to be aware that this may happen.

Relation to `slice()` object

The slicing operator `[]` is actually being used in the above code with a `slice()` object using the `:` notation (which is only valid within `[]`), i.e.:

```
a[start:stop:step]
```

is equivalent to:

```
a[slice(start, stop, step)]
```

Slice objects also behave slightly differently depending on the number of arguments, similarly to `range()`, i.e. both `slice(stop)` and `slice(start, stop[, step])` are supported. To skip specifying a given argument, one might use `None`, so that e.g. `a[start:]` is equivalent to `a[slice(start, None)]` or `a[::-1]` is equivalent to `a[slice(None, None, -1)]`.

While the `:`-based notation is very helpful for simple slicing, the explicit use of `slice()` objects simplifies the programmatic generation of slicing.

edited Feb 24 at 19:27

answered Feb 3 '09 at 22:48

Greg Hewgill
758k ● 164 ● 1065 ● 1210

120 Slicing builtin types returns a copy but that's not universal. Notably, slicing NumPy arrays returns a view that shares memory with the original. – Beni Cherniavsky-Paskin Sep 23 '13 at 0:13

41 This is a beautiful answer with the votes to prove it, but it misses one thing: you can substitute `None` for any of the empty spaces. For example `[None:None]` makes a whole copy. This is useful when you need to specify the end of the range using a variable and need to include the last item. – Mark Ransom Jan 16 '19 at 18:49

5 What really annoys me is that python says that when you don't set the start and the end, they default to 0 and the length of sequence. So, in theory, when you use "abcdef"[::-1] it should be transformed to "abcdef"[0:6:-1], but these two expressions does not get the same output. I feel that something is missing in python documentation since the creation of the language. – axell13 Jun 30 '19 at 14:00

5 And I know that "abcdef"[::-1] is transformed to "abcdef"[6:-7:-1], so, the best way to explain would be: let *len* be the length of the sequence. **If step is positive**, the defaults for start and end are 0 and *len*. **Else if step is negative**, the defaults for start and end are *len* and *-len* - 1. – axell13 Jun 30 '19 at 14:22

4 Since stackoverflow.com/questions/39241529/… is marked dupe of this, would it make sense to add a section with what `del` does w.r.t. slice notation. In particular, `del arr[:]` isn't immediately obvious ("arr[:] makes a copy, so does del delete that copy???" etc.) – khazhyk Jul 26 '19 at 1:32

The [Python tutorial] talks about it (scroll down a bit until you get to the part about slicing).

535

The ASCII art diagram is helpful too for remembering how slices work:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

> One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*.

edited Sep 18 '17 at 11:02

kenorb
97.2k ● 43 ● 513 ● 548

answered Feb 3 '09 at 22:49

Hans Nowak
5,864 ● 1 ● 16 ● 13

9 This suggestion works for positive stride, but does not for a negative stride. From the diagram, I expect `a[-4,-6,-1]` to be `yP` but it is `ty`. What always work is to think in characters or slots and use indexing as a half-open interval -- right-open if positive stride, left-open if negative stride. – aguadopd May 27 '19 at 20:05

409

Enumerating the possibilities allowed by the grammar:

```
>>> seq[:]                # [seq[0],   seq[1],          ..., seq[-1]    ]
>>> seq[low:]             # [seq[low], seq[low+1],      ..., seq[-1]    ]
>>> seq[:high]            # [seq[0],   seq[1],          ..., seq[high-1]]
>>> seq[low:high]         # [seq[low], seq[low+1],      ..., seq[high-1]]
>>> seq[::stride]         # [seq[0],   seq[stride],     ..., seq[-1]    ]
>>> seq[low::stride]      # [seq[low], seq[low+stride], ..., seq[-1]    ]
>>> seq[:high:stride]     # [seq[0],   seq[stride],     ..., seq[high-1]]
>>> seq[low:high:stride]  # [seq[low], seq[low+stride], ..., seq[high-1]]
```

Of course, if `(high-low)%stride != 0`, then the end point will be a little lower than `high-1`.

If `stride` is negative, the ordering is changed a bit since we're counting down:

```
>>> seq[::-stride]        # [seq[-1],   seq[-1-stride],   ..., seq[0]    ]
>>> seq[high::-stride]    # [seq[high], seq[high-stride], ..., seq[0]    ]
>>> seq[:low:-stride]     # [seq[-1],   seq[-1-stride],   ..., seq[low+1]]
>>> seq[high:low:-stride] # [seq[high], seq[high-stride], ..., seq[low+1]]
```

Extended slicing (with commas and ellipses) are mostly used only by special data structures (like NumPy); the basic sequences don't support them.

```
>>> class slicee:
...     def __getitem__(self, item):
...         return repr(item)
...
```

```
>>> slicee()[0, 1:2, ::5, ...]
'(0, slice(1, 2, None), slice(None, None, 5), Ellipsis)'
```

edited May 7 '19 at 12:16
Georgy
3,937 ● 5 ● 30 ● 41

answered Feb 3 '09 at 23:08
ephemient
169k ● 32 ● 245 ● 366

318

The answers above don't discuss slice assignment. To understand slice assignment, it's helpful to add another concept to the ASCII art:

```
               +---+---+---+---+---+---+
               | P | y | t | h | o | n |
               +---+---+---+---+---+---+
Slice position: 0   1   2   3   4   5   6
Index position:   0   1   2   3   4   5

>>> p = ['P','y','t','h','o','n']
# Why the two sets of numbers:
# indexing gives items, not lists
>>> p[0]
 'P'
>>> p[5]
 'n'

# Slicing gives lists
>>> p[0:1]
 ['P']
>>> p[0:2]
 ['P','y']
```

One heuristic is, for a slice from zero to n, think: "zero is the beginning, start at the beginning and take n items in a list".

```
>>> p[5] # the last of six items, indexed from zero
 'n'
>>> p[0:5] # does NOT include the last item!
 ['P','y','t','h','o']
>>> p[0:6] # not p[0:5]!!!
 ['P','y','t','h','o','n']
```

Another heuristic is, "for any slice, replace the start by zero, apply the previous heuristic to get the end of the list, then count the first number back up to chop items off the beginning"

```
>>> p[0:4] # Start at the beginning and count out 4 items
 ['P','y','t','h']
>>> p[1:4] # Take one item off the front
 ['y','t','h']
>>> p[2:4] # Take two items off the front
 ['t','h']
# etc.
```

The first rule of slice assignment is that since slicing *returns* a list, slice assignment *requires* a list (or other iterable):

```
>>> p[2:3]
 ['t']
>>> p[2:3] = ['T']
>>> p
 ['P','y','T','h','o','n']
>>> p[2:3] = 't'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only assign an iterable
```

The second rule of slice assignment, which you can also see above, is that whatever portion of the list is returned by slice indexing, that's the same portion that is changed by slice assignment:

```
>>> p[2:4]
 ['T','h']
>>> p[2:4] = ['t','r']
>>> p
 ['P','y','t','r','o','n']
```

The third rule of slice assignment is, the assigned list (iterable) doesn't have to have the same length; the indexed slice is simply sliced out and replaced en masse by whatever is being assigned:

```
>>> p = ['P','y','t','h','o','n'] # Start over
>>> p[2:4] = ['s','p','a','m']
>>> p
 ['P','y','s','p','a','m','o','n']
```

The trickiest part to get used to is assignment to empty slices. Using heuristic 1 and 2 it's easy to get your head around *indexing* an empty slice:

```
>>> p = ['P','y','t','h','o','n']
>>> p[0:4]
 ['P','y','t','h']
>>> p[1:4]
 ['y','t','h']
>>> p[2:4]
 ['t','h']
>>> p[3:4]
 ['h']
>>> p[4:4]
 []
```

And then once you've seen that, slice assignment to the empty slice makes sense too:

```
>>> p = ['P','y','t','h','o','n']
>>> p[2:4] = ['x','y'] # Assigned list is same length as slice
>>> p
 ['P','y','x','y','o','n'] # Result is same length
>>> p = ['P','y','t','h','o','n']
>>> p[3:4] = ['x','y'] # Assigned list is longer than slice
>>> p
 ['P','y','t','x','y','o','n'] # The result is longer
>>> p = ['P','y','t','h','o','n']
>>> p[4:4] = ['x','y']
```

```
>>> p
 ['P','y','t','h','x','y','o','n'] # The result is longer still
```

Note that, since we are not changing the second number of the slice (4), the inserted items always stack right up against the 'o', even when we're assigning to the empty slice. So the position for the empty slice assignment is the logical extension of the positions for the non-empty slice assignments.

Backing up a little bit, what happens when you keep going with our procession of counting up the slice beginning?

```
>>> p = ['P','y','t','h','o','n']
>>> p[0:4]
 ['P','y','t','h']
>>> p[1:4]
 ['y','t','h']
>>> p[2:4]
 ['t','h']
>>> p[3:4]
 ['h']
>>> p[4:4]
 []
>>> p[5:4]
 []
>>> p[6:4]
 []
```

With slicing, once you're done, you're done; it doesn't start slicing backwards. In Python you don't get negative strides unless you explicitly ask for them by using a negative number.

```
>>> p[5:3:-1]
 ['n','o']
```

There are some weird consequences to the "once you're done, you're done" rule:

```
>>> p[4:4]
 []
>>> p[5:4]
 []
>>> p[6:4]
 []
>>> p[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

In fact, compared to indexing, Python slicing is bizarrely error-proof:

```
>>> p[100:200]
 []
>>> p[int(2e99):int(1e99)]
 []
```

This can come in handy sometimes, but it can also lead to somewhat strange behavior:

```
>>> p
 ['P', 'y', 't', 'h', 'o', 'n']
>>> p[int(2e99):int(1e99)] = ['p','o','w','e','r']
>>> p
 ['P', 'y', 't', 'h', 'o', 'n', 'p', 'o', 'w', 'e', 'r']
```

Depending on your application, that might... or might not... be what you were hoping for there!

Below is the text of my original answer. It has been useful to many people, so I didn't want to delete it.

```
>>> r=[1,2,3,4]
>>> r[1:1]
[]
>>> r[1:1]=[9,8]
>>> r
[1, 9, 8, 2, 3, 4]
>>> r[1:1]=['blah']
>>> r
[1, 'blah', 9, 8, 2, 3, 4]
```

This may also clarify the difference between slicing and indexing.

edited Jan 2 '19 at 16:44

Peter Mortensen
25.7k ● 21 ● 90 ● 118

answered Jan 18 '11 at 21:37

David M. Perlman
3,591 ● 1 ● 13 ● 11

244

Explain Python's slice notation

In short, the colons (`:`) in subscript notation (`subscriptable[subscriptarg]`) make slice notation - which has the optional arguments, `start` , `stop` , `step` :

```
sliceable[start:stop:step]
```

Python slicing is a computationally fast way to methodically access parts of your data. In my opinion, to be even an intermediate Python programmer, it's one aspect of the language that it is necessary to be familiar with.

Important Definitions

To begin with, let's define a few terms:

start: the beginning index of the slice, it will include the element at this index unless it is the same as *stop*, defaults to 0, i.e. the first index. If it's negative, it means to start `n` items from the end.

stop: the ending index of the slice, it does *not* include the element at this index, defaults to length of the sequence being sliced, that is, up to and including the end.

> **step:** the amount by which the index increases, defaults to 1. If it's negative, you're slicing over the iterable in reverse.

How Indexing Works

You can make any of these positive or negative numbers. The meaning of the positive numbers is straightforward, but for negative numbers, just like indexes in Python, you count backwards from the end for the *start* and *stop*, and for the *step*, you simply decrement your index. This example is from the documentation's tutorial, but I've modified it slightly to indicate which item in a sequence each index references:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
  0   1   2   3   4   5
 -6  -5  -4  -3  -2  -1
```

How Slicing Works

To use slice notation with a sequence that supports it, you must include at least one colon in the square brackets that follow the sequence (which actually implement the `__getitem__` method of the sequence, according to the Python data model.)

Slice notation works like this:

```
sequence[start:stop:step]
```

And recall that there are defaults for *start*, *stop*, and *step*, so to access the defaults, simply leave out the argument.

Slice notation to get the last nine elements from a list (or any other sequence that supports it, like a string) would look like this:

```
my_list[-9:]
```

When I see this, I read the part in the brackets as "9th from the end, to the end." (Actually, I abbreviate it mentally as "-9, on")

Explanation:

The full notation is

```
my_list[-9:None:None]
```

and to substitute the defaults (actually when `step` is negative, `stop`'s default is `-len(my_list) - 1`, so `None` for stop really just means it goes to whichever end step takes it to):

```
my_list[-9:len(my_list):1]
```

The **colon**, `:`, is what tells Python you're giving it a slice and not a regular index. That's why the idiomatic way of making a shallow copy of lists in Python 2 is

```
list_copy = sequence[:]
```

And clearing them is with:

```
del my_list[:]
```

(Python 3 gets a `list.copy` and `list.clear` method.)

When `step` is negative, the defaults for `start` and `stop` change

By default, when the `step` argument is empty (or `None`), it is assigned to `+1`.

But you can pass in a negative integer, and the list (or most other standard slicables) will be sliced from the end to the beginning.

Thus a negative slice will change the defaults for `start` and `stop`!

Confirming this in the source

I like to encourage users to read the source as well as the documentation. The source code for slice objects and this logic is found here. First we determine if `step` is negative:

```
step_is_negative = step_sign < 0;
```

If so, the lower bound is `-1` meaning we slice all the way up to and including the beginning, and the upper bound is the length minus 1, meaning we start at the end. (Note that the semantics of this `-1` is *different* from a `-1` that users may pass indexes in Python indicating the last item.)

```
if (step_is_negative) {
    lower = PyLong_FromLong(-1L);
    if (lower == NULL)
        goto error;

    upper = PyNumber_Add(length, lower);
    if (upper == NULL)
        goto error;
}
```

Otherwise `step` is positive, and the lower bound will be zero and the upper bound (which we go up to but not including) the length of the sliced list.

```
else {
    lower = _PyLong_Zero;
    Py_INCREF(lower);
    upper = length;
```

```
    Py_INCREF(upper);
}
```

Then, we may need to apply the defaults for `start` and `stop` - the default then for `start` is calculated as the upper bound when `step` is negative:

```
if (self->start == Py_None) {
    start = step_is_negative ? upper : lower;
    Py_INCREF(start);
}
```

and `stop`, the lower bound:

```
if (self->stop == Py_None) {
    stop = step_is_negative ? lower : upper;
    Py_INCREF(stop);
}
```

Give your slices a descriptive name!

You may find it useful to separate forming the slice from passing it to the `list.__getitem__` method (that's what the square brackets do). Even if you're not new to it, it keeps your code more readable so that others that may have to read your code can more readily understand what you're doing.

However, you can't just assign some integers separated by colons to a variable. You need to use the slice object:

```
last_nine_slice = slice(-9, None)
```

The second argument, `None`, is required, so that the first argument is interpreted as the `start` argument otherwise it would be the `stop` argument.

You can then pass the slice object to your sequence:

```
>>> list(range(100))[last_nine_slice]
[91, 92, 93, 94, 95, 96, 97, 98, 99]
```

It's interesting that ranges also take slices:

```
>>> range(100)[last_nine_slice]
range(91, 100)
```

Memory Considerations:

Since slices of Python lists create new objects in memory, another important function to be aware of is `itertools.islice`. Typically you'll want to iterate over a slice, not just have it created statically in memory. `islice` is perfect for this. A caveat, it doesn't support negative arguments to `start`, `stop`, or `step`, so if that's an issue you may need to calculate indices or reverse the iterable in advance.

```
length = 100
last_nine_iter = itertools.islice(list(range(length)), length-9, None, 1)
list_last_nine = list(last_nine_iter)
```

and now:

```
>>> list_last_nine
[91, 92, 93, 94, 95, 96, 97, 98, 99]
```

The fact that list slices make a copy is a feature of lists themselves. If you're slicing advanced objects like a Pandas DataFrame, it may return a view on the original, and not a copy.

edited Jun 13 '18 at 20:35

answered Jul 12 '14 at 13:19

Aaron Hall ♦

240k ● 64 ● 344 ● 293

145

And a couple of things that weren't immediately obvious to me when I first saw the slicing syntax:

```
>>> x = [1,2,3,4,5,6]
>>> x[::-1]
[6,5,4,3,2,1]
```

Easy way to reverse sequences!

And if you wanted, for some reason, every second item in the reversed sequence:

```
>>> x = [1,2,3,4,5,6]
>>> x[::-2]
[6,4,2]
```

answered Feb 3 '09 at 23:15

Dana

26.4k ● 16 ● 57 ● 72

100

In Python 2.7

Slicing in Python

```
[a:b:c]

len = length of string, tuple or list

c -- default is +1. The sign of c indicates forward or backward, absolute value of c
```

```
indicates steps. Default is forward with step size 1. Positive means forward, negative
means backward.

a --  When c is positive or blank, default is 0. When c is negative, default is -1.

b --  When c is positive or blank, default is len. When c is negative, default is -
(len+1).
```

Understanding index assignment is very important.

```
In forward direction, starts at 0 and ends at len-1

In backward direction, starts at -1 and ends at -len
```

When you say [a:b:c], you are saying depending on the sign of c (forward or backward), start at a and end at b (excluding element at bth index). Use the indexing rule above and remember you will only find elements in this range:

```
-len, -len+1, -len+2, ..., 0, 1, 2,3,4 , len -1
```

But this range continues in both directions infinitely:

```
...,-len -2 ,-len-1,-len, -len+1, -len+2, ..., 0, 1, 2,3,4 , len -1, len, len +1, len+2
, ....
```

For example:

```
            0    1    2   3    4   5   6   7   8   9   10   11
            a    s    t   r    i   n   g
   -9  -8  -7   -6   -5  -4   -3  -2  -1
```

If your choice of a, b, and c allows overlap with the range above as you traverse using rules for a,b,c above you will either get a list with elements (touched during traversal) or you will get an empty list.

One last thing: if a and b are equal, then also you get an empty list:

```
>>> l1
[2, 3, 4]

>>> l1[:]
[2, 3, 4]

>>> l1[::-1] # a default is -1 , b default is -(len+1)
[4, 3, 2]

>>> l1[:-4:-1] # a default is -1
[4, 3, 2]

>>> l1[:-3:-1] # a default is -1
[4, 3]

>>> l1[::] # c default is +1, so a default is 0, b default is len
[2, 3, 4]
```

```
>>> l1[::-1] # c is -1 , so a default is -1 and b default is -(len+1)
[4, 3, 2]


>>> l1[-100:-200:-1] # Interesting
[]

>>> l1[-1:-200:-1] # Interesting
[4, 3, 2]


>>> l1[-1:-1:1]
[]


>>> l1[-1:5:1] # Interesting
[4]


>>> l1[1:-7:1]
[]

>>> l1[1:-7:-1] # Interesting
[3, 2]

>>> l1[:-2:-2] # a default is -1, stop(b) at -2 , step(c) by 2 in reverse direction
[4]
```

edited Jul 10 '17 at 16:59

answered Oct 22 '12 at 5:33

Ankur Agarwal

17.7k ● 30 ● 98 ● 163

2 another one interesting example: `a = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]; a[:-2:-2]` which results to `[9]` – Deviacium Jul 10 '17 at 13:59

96

Found this great table at http://wiki.python.org/moin/MovingToPythonFromOtherLanguages

```
Python indexes and slices for a six-element list.
Indexes enumerate the elements, slices enumerate the spaces between the elements.

Index from rear:    -6  -5  -4  -3  -2  -1      a=[0,1,2,3,4,5]    a[1:]==[1,2,3,4,5]
Index from front:    0   1   2   3   4   5      len(a)==6          a[:5]==[0,1,2,3,4]
                  +---+---+---+---+---+---+     a[0]==0           a[:-2]==[0,1,2,3]
                  | a | b | c | d | e | f |     a[5]==5           a[1:2]==[1]
                  +---+---+---+---+---+---+     a[-1]==5          a[1:-1]==[1,2,3,4]
Slice from front: :   1   2   3   4   5   :     a[-2]==4
Slice from rear:  :  -5  -4  -3  -2  -1   :
                                               b=a[:]
                                               b==[0,1,2,3,4,5] (shallow copy of a)
```

answered Sep 6 '11 at 6:50

AdrianoFerrari

1,968 ● 17 ● 14

After using it a bit I realise that the simplest description is that it is exactly the same as the

65

arguments in a `for` loop...

```
(from:to:step)
```

Any of them are optional:

```
(:to:step)
(from::step)
(from:to)
```

Then the negative indexing just needs you to add the length of the string to the negative indices to understand it.

This works for me anyway...

edited Jan 2 '19 at 16:40

Peter Mortensen
25.7k • 21 • 90 • 118

answered Feb 19 '09 at 20:52

Simon
63.8k • 24 • 80 • 116

52

I find it easier to remember how it works, and then I can figure out any specific start/stop/step combination.

It's instructive to understand `range()` first:

```
def range(start=0, stop, step=1):  # Illegal syntax, but that's the effect
    i = start
    while (i < stop if step > 0 else i > stop):
        yield i
        i += step
```

Begin from `start`, increment by `step`, do not reach `stop`. Very simple.

The thing to remember about negative step is that `stop` is always the excluded end, whether it's higher or lower. If you want same slice in opposite order, it's much cleaner to do the reversal separately: e.g. `'abcde'[1:-2][::-1]` slices off one char from left, two from right, then reverses. (See also `reversed()`.)

Sequence slicing is same, except it first normalizes negative indexes, and it can never go outside the sequence:

TODO: The code below had a bug with "never go outside the sequence" when abs(step)>1; I *think* I patched it to be correct, but it's hard to understand.

```
def this_is_how_slicing_works(seq, start=None, stop=None, step=1):
    if start is None:
        start = (0 if step > 0 else len(seq)-1)
    elif start < 0:
        start += len(seq)
    if not 0 <= start < len(seq):  # clip if still outside bounds
        start = (0 if step > 0 else len(seq)-1)
    if stop is None:
        stop = (len(seq) if step > 0 else -1)  # really -1, not last element
```

```
    elif stop < 0:
        stop += len(seq)
    for i in range(start, stop, step):
        if 0 <= i < len(seq):
            yield seq[i]
```

Don't worry about the `is None` details - just remember that omitting `start` and/or `stop` always does the right thing to give you the whole sequence.

Normalizing negative indexes first allows start and/or stop to be counted from the end independently: `'abcde'[1:-2] == 'abcde'[1:3] == 'bc'` despite `range(1,-2) == []`. The normalization is sometimes thought of as "modulo the length", but note it adds the length just once: e.g. `'abcde'[-53:42]` is just the whole string.

edited Jan 2 '19 at 16:46

Peter Mortensen
25.7k ● 21 ● 90 ● 118

answered Mar 29 '12 at 10:15

Beni Cherniavsky-Paskin
7,431 ● 35 ● 50

3 The `this_is_how_slicing_works` is not the same as python slice. E.G. `[0, 1, 2][-5:3:3]` will get [0] in python, but `list(this_is_how_slicing_works([0, 1, 2], -5, 3, 3))` get [1]. – Eastsun Oct 29 '16 at 12:56

40

I use the "an index points between elements" method of thinking about it myself, but one way of describing it which sometimes helps others get it is this:

```
mylist[X:Y]
```

X is the index of the first element you want.
Y is the index of the first element you *don't* want.

answered Feb 6 '09 at 21:16

Steve Losh
18.2k ● 2 ● 47 ● 44

40

```
Index:
      ------------>
  0   1   2   3   4
+---+---+---+---+---+
| a | b | c | d | e |
+---+---+---+---+---+
  0  -4  -3  -2  -1
     <------------

Slice:
    <---------------|
|--------------->
:   1   2   3   4   :
+---+---+---+---+---+
| a | b | c | d | e |
+---+---+---+---+---+
:  -4  -3  -2  -1   :
```

```
|--------------->
    <---------------|
```

I hope this will help you to model the list in Python.

Reference: http://wiki.python.org/moin/MovingToPythonFromOtherLanguages

edited Feb 11 '17 at 19:56

Peter Mortensen
25.7k ● 21 ● 90 ● 118

answered Feb 4 '13 at 7:20

xiaoyu
6,792 ● 1 ● 13 ● 12

38

Python slicing notation:

```
a[start:end:step]
```

- For `start` and `end`, negative values are interpreted as being relative to the end of the sequence.
- Positive indices for `end` indicate the position *after* the last element to be included.
- Blank values are defaulted as follows: `[+0:-0:1]`.
- Using a negative step reverses the interpretation of `start` and `end`

The notation extends to (numpy) matrices and multidimensional arrays. For example, to slice entire columns you can use:

```
m[::,0:2:] ## slice the first two columns
```

Slices hold references, not copies, of the array elements. If you want to make a separate copy an array, you can use `deepcopy()`.

edited May 23 '17 at 12:34

Community ♦
1 ● 1

answered Apr 28 '13 at 19:49

nobar
34.2k ● 11 ● 106 ● 121

34

You can also use slice assignment to remove one or more elements from a list:

```
r = [1, 'blah', 9, 8, 2, 3, 4]
>>> r[1:4] = []
>>> r
[1, 2, 3, 4]
```

edited Apr 19 '13 at 16:28

answered Apr 5 '13 at 1:59

dansalmo
9,525 ● 4 ● 46 ● 47

This is just for some extra info... Consider the list below

33

```
>>> l=[12,23,345,456,67,7,945,467]
```

Few other tricks for reversing the list:

```
>>> l[len(l):-len(l)-1:-1]
[467, 945, 7, 67, 456, 345, 23, 12]

>>> l[:-len(l)-1:-1]
[467, 945, 7, 67, 456, 345, 23, 12]

>>> l[len(l)::-1]
[467, 945, 7, 67, 456, 345, 23, 12]

>>> l[::-1]
[467, 945, 7, 67, 456, 345, 23, 12]

>>> l[-1:-len(l)-1:-1]
[467, 945, 7, 67, 456, 345, 23, 12]
```

edited May 8 '19 at 8:35
Georgy
3,937 ● 5 ● 30 ● 41

answered Mar 22 '12 at 17:20

Arindam Roychowdhury
2,952 ● 5 ● 35 ● 40

33

This is how I teach slices to newbies:

Understanding the difference between indexing and slicing:

Wiki Python has this amazing picture which clearly distinguishes indexing and slicing.

```
Index from rear:    -6  -5  -4  -3  -2  -1
Index from front:    0   1   2   3   4   5
                   +---+---+---+---+---+---+
                   | a | b | c | d | e | f |
                   +---+---+---+---+---+---+
Slice from front:  :   1   2   3   4   5   :
Slice from rear:   :  -5  -4  -3  -2  -1   :
```

It is a list with six elements in it. To understand slicing better, consider that list as a set of six boxes placed together. Each box has an alphabet in it.

Indexing is like dealing with the contents of box. You can check contents of any box. But you can't check the contents of multiple boxes at once. You can even replace the contents of the box. But you can't place two balls in one box or replace two balls at a time.

```
In [122]: alpha = ['a', 'b', 'c', 'd', 'e', 'f']

In [123]: alpha
Out[123]: ['a', 'b', 'c', 'd', 'e', 'f']

In [124]: alpha[0]
Out[124]: 'a'
```

```
In [127]: alpha[0] = 'A'

In [128]: alpha
Out[128]: ['A', 'b', 'c', 'd', 'e', 'f']

In [129]: alpha[0,1]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-129-c7eb16585371> in <module>()
----> 1 alpha[0,1]

TypeError: list indices must be integers, not tuple
```

Slicing is like dealing with boxes themselves. You can pick up the first box and place it on another table. To pick up the box, all you need to know is the position of beginning and ending of the box.

You can even pick up the first three boxes or the last two boxes or all boxes between 1 and 4. So, you can pick any set of boxes if you know the beginning and ending. These positions are called start and stop positions.

The interesting thing is that you can replace multiple boxes at once. Also you can place multiple boxes wherever you like.

```
In [130]: alpha[0:1]
Out[130]: ['A']

In [131]: alpha[0:1] = 'a'

In [132]: alpha
Out[132]: ['a', 'b', 'c', 'd', 'e', 'f']

In [133]: alpha[0:2] = ['A', 'B']

In [134]: alpha
Out[134]: ['A', 'B', 'c', 'd', 'e', 'f']

In [135]: alpha[2:2] = ['x', 'xx']

In [136]: alpha
Out[136]: ['A', 'B', 'x', 'xx', 'c', 'd', 'e', 'f']
```

Slicing With Step:

Till now you have picked boxes continuously. But sometimes you need to pick up discretely. For example, you can pick up every second box. You can even pick up every third box from the end. This value is called step size. This represents the gap between your successive pickups. The step size should be positive if You are picking boxes from the beginning to end and vice versa.

```
In [137]: alpha = ['a', 'b', 'c', 'd', 'e', 'f']

In [142]: alpha[1:5:2]
Out[142]: ['b', 'd']

In [143]: alpha[-1:-5:-2]
Out[143]: ['f', 'd']

In [144]: alpha[1:5:-2]
Out[144]: []
```

```
In [145]: alpha[-1:-5:2]
Out[145]: []
```

How Python Figures Out Missing Parameters:

When slicing, if you leave out any parameter, Python tries to figure it out automatically.

If you check the source code of CPython, you will find a function called PySlice_GetIndicesEx() which figures out indices to a slice for any given parameters. Here is the logical equivalent code in Python.

This function takes a Python object and optional parameters for slicing and returns the start, stop, step, and slice length for the requested slice.

```
def py_slice_get_indices_ex(obj, start=None, stop=None, step=None):

    length = len(obj)

    if step is None:
        step = 1
    if step == 0:
        raise Exception("Step cannot be zero.")

    if start is None:
        start = 0 if step > 0 else length - 1
    else:
        if start < 0:
            start += length
        if start < 0:
            start = 0 if step > 0 else -1
        if start >= length:
            start = length if step > 0 else length - 1

    if stop is None:
        stop = length if step > 0 else -1
    else:
        if stop < 0:
            stop += length
        if stop < 0:
            stop = 0 if step > 0 else -1
        if stop >= length:
            stop = length if step > 0 else length - 1

    if (step < 0 and stop >= start) or (step > 0 and start >= stop):
        slice_length = 0
    elif step < 0:
        slice_length = (stop - start + 1)/(step) + 1
    else:
        slice_length = (stop - start - 1)/(step) + 1

    return (start, stop, step, slice_length)
```

This is the intelligence that is present behind slices. Since Python has an built-in function called slice, you can pass some parameters and check how smartly it calculates missing parameters.

```
In [21]: alpha = ['a', 'b', 'c', 'd', 'e', 'f']

In [22]: s = slice(None, None, None)

In [23]: s
```

```
Out[23]: slice(None, None, None)

In [24]: s.indices(len(alpha))
Out[24]: (0, 6, 1)

In [25]: range(*s.indices(len(alpha)))
Out[25]: [0, 1, 2, 3, 4, 5]

In [26]: s = slice(None, None, -1)

In [27]: range(*s.indices(len(alpha)))
Out[27]: [5, 4, 3, 2, 1, 0]

In [28]: s = slice(None, 3, -1)

In [29]: range(*s.indices(len(alpha)))
Out[29]: [5, 4]
```

Note: This post was originally written in my blog, *The Intelligence Behind Python Slices*.

edited Sep 26 '19 at 7:58

Peter Mortensen

25.7k ● 21 ● 90 ● 118

answered Mar 24 '15 at 16:08

ChillarAnand

19.7k ● 6 ● 90 ● 106

29

As a general rule, writing code with a lot of hardcoded index values leads to a readability and maintenance mess. For example, if you come back to the code a year later, you'll look at it and wonder what you were thinking when you wrote it. The solution shown is simply a way of more clearly stating what your code is actually doing. In general, the built-in slice() creates a slice object that can be used anywhere a slice is allowed. For example:

```
>>> items = [0, 1, 2, 3, 4, 5, 6]
>>> a = slice(2, 4)
>>> items[2:4]
[2, 3]
>>> items[a]
[2, 3]
>>> items[a] = [10,11]
>>> items
[0, 1, 10, 11, 4, 5, 6]
>>> del items[a]
>>> items
[0, 1, 4, 5, 6]
```

If you have a slice instance s, you can get more information about it by looking at its s.start, s.stop, and s.step attributes, respectively. For example:

```
>>> a = slice(10, 50, 2)
>>> a.start
10
>>> a.stop
50
>>> a.step
2
>>>
```

answered Dec 7 '13 at 16:52

Python_Dude
497 ● 5 ● 11

25

1. Slice Notation

To make it simple, remember **slice has only one form：**

```
s[start:end:step]
```

and here is how it works:

- `s`: an object that can be sliced
- `start`: first index to start iteration
- `end`: last index, **NOTE that `end` index will not be included in the resulted slice**
- `step`: pick element every `step` index

Another import thing: **all `start`, `end`, `step` can be omitted!** And if they are omitted, their default value will be used: `0`, `len(s)`, `1` accordingly.

So possible variations are:

```
# Mostly used variations
s[start:end]
s[start:]
s[:end]

# Step-related variations
s[:end:step]
s[start::step]
s[::step]

# Make a copy
s[:]
```

NOTE: If `start >= end` (considering only when `step>0`), Python will return a empty slice `[]`.

2. Pitfalls

The above part explains the core features on how slice works, and it will work on most occasions. However, there can be pitfalls you should watch out, and this part explains them.

Negative indexes

The very first thing that confuses Python learners is that **an index can be negative!** Don't panic: **a negative index means count backwards.**

For example:

```
s[-5:]    # Start at the 5th index from the end of array,
          # thus returning the last 5 elements.
```

```
s[:-5]    # Start at index 0, and end until the 5th index from end of array,
          # thus returning s[0:len(s)-5].
```

Negative step

Making things more confusing is that `step` **can be negative too!**

A negative step means iterate the array backwards: from the end to start, with the end index included, and the start index excluded from the result.

NOTE: when step is negative, the default value for `start` is `len(s)` (while `end` does not equal to `0`, because `s[::-1]` contains `s[0]`). For example:

```
s[::-1]            # Reversed slice
s[len(s)::-1]      # The same as above, reversed slice
s[0:len(s):-1]     # Empty list
```

Out of range error?

Be surprised: **slice does not raise an IndexError when the index is out of range!**

If the index is out of range, Python will try its best to set the index to `0` or `len(s)` according to the situation. For example:

```
s[:len(s)+5]      # The same as s[:len(s)]
s[-len(s)-5::]    # The same as s[0:]
s[len(s)+5::-1]   # The same as s[len(s)::-1], and the same as s[::-1]
```

3. Examples

Let's finish this answer with examples, explaining everything we have discussed:

```
# Create our array for demonstration
In [1]: s = [i for i in range(10)]

In [2]: s
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: s[2:]   # From index 2 to last index
Out[3]: [2, 3, 4, 5, 6, 7, 8, 9]

In [4]: s[:8]   # From index 0 up to index 8
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7]

In [5]: s[4:7]  # From index 4 (included) up to index 7(excluded)
Out[5]: [4, 5, 6]

In [6]: s[:-2]  # Up to second last index (negative index)
Out[6]: [0, 1, 2, 3, 4, 5, 6, 7]

In [7]: s[-2:]  # From second last index (negative index)
Out[7]: [8, 9]

In [8]: s[::-1] # From last to first in reverse order (negative step)
Out[8]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
In [9]: s[::-2] # All odd numbers in reversed order
Out[9]: [9, 7, 5, 3, 1]

In [11]: s[-2::-2] # All even numbers in reversed order
Out[11]: [8, 6, 4, 2, 0]

In [12]: s[3:15]   # End is out of range, and Python will set it to len(s).
Out[12]: [3, 4, 5, 6, 7, 8, 9]

In [14]: s[5:1]    # Start > end; return empty list
Out[14]: []

In [15]: s[11]     # Access index 11 (greater than len(s)) will raise an IndexError
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-15-79ffc22473a3> in <module>()
----> 1 s[11]

IndexError: list index out of range
```

edited Sep 26 '19 at 8:04 — Peter Mortensen — 25.7k ● 21 ● 90 ● 118

answered Jan 9 '17 at 12:52 — cizixs — 8,103 ● 5 ● 40 ● 55

23

The previous answers don't discuss multi-dimensional array slicing which is possible using the famous NumPy package:

Slicing can also be applied to multi-dimensional arrays.

```
# Here, a is a NumPy array

>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> a[:2, 0:3:2]
array([[1, 3],
       [5, 7]])
```

The "`:2`" before the comma operates on the first dimension and the "`0:3:2`" after the comma operates on the second dimension.

edited Sep 26 '19 at 8:08 — Peter Mortensen — 25.7k ● 21 ● 90 ● 118

answered Mar 1 '17 at 2:31 — Statham — 2,730 ● 1 ● 22 ● 37

4 Just a friendly reminder that you cannot do this on Python `list` but only on `array` in Numpy – Mars Lee Jul 26 '19 at 21:46

15

```
#!/usr/bin/env python

def slicegraphical(s, lista):
```

```
    if len(s) > 9:
        print """Enter a string of maximum 9 characters,
    so the printig would looki nice"""
        return 0;
    # print " ",
    print '  '+'+---' * len(s) +'+'
    print ' ',
    for letter in s:
        print '| {}'.format(letter),
    print '|'
    print " ",; print '+---' * len(s) +'+'

    print " ",
    for letter in range(len(s) +1):
        print '{}  '.format(letter),
    print ""
    for letter in range(-1*(len(s)), 0):
        print ' {}'.format(letter),
    print ''
    print ''


    for triada in lista:
        if len(triada) == 3:
            if triada[0]==None and triada[1] == None and triada[2] == None:
                # 000
                print s+'[   :   :   ]' +' = ', s[triada[0]:triada[1]:triada[2]]
            elif triada[0] == None and triada[1] == None and triada[2] != None:
                # 001
                print s+'[   :   :{0:2d} ]'.format(triada[2], '','') +' = ',
s[triada[0]:triada[1]:triada[2]]
            elif triada[0] == None and triada[1] != None and triada[2] == None:
                # 010
                print s+'[   :{0:2d} :   ]'.format(triada[1]) +' = ',
s[triada[0]:triada[1]:triada[2]]
            elif triada[0] == None and triada[1] != None and triada[2] != None:
                # 011
                print s+'[   :{0:2d} :{1:2d} ]'.format(triada[1], triada[2]) +' = ',
s[triada[0]:triada[1]:triada[2]]
            elif triada[0] != None and triada[1] == None and triada[2] == None:
                # 100
                print s+'[{0:2d} :   :   ]'.format(triada[0]) +' = ',
s[triada[0]:triada[1]:triada[2]]
            elif triada[0] != None and triada[1] == None and triada[2] != None:
                # 101
                print s+'[{0:2d} :   :{1:2d} ]'.format(triada[0], triada[2]) +' = ',
s[triada[0]:triada[1]:triada[2]]
            elif triada[0] != None and triada[1] != None and triada[2] == None:
                # 110
                print s+'[{0:2d} :{1:2d} :   ]'.format(triada[0], triada[1]) +' = ',
s[triada[0]:triada[1]:triada[2]]
            elif triada[0] != None and triada[1] != None and triada[2] != None:
                # 111
                print s+'[{0:2d} :{1:2d} :{2:2d} ]'.format(triada[0], triada[1],
triada[2]) +' = ', s[triada[0]:triada[1]:triada[2]]

        elif len(triada) == 2:
            if triada[0] == None and triada[1] == None:
                # 00
                print s+'[   :   ]    ' + ' = ', s[triada[0]:triada[1]]
            elif triada[0] == None and triada[1] != None:
                # 01
                print s+'[   :{0:2d} ]    '.format(triada[1]) + ' = ',
s[triada[0]:triada[1]]
            elif triada[0] != None and triada[1] == None:
```

```
                # 10
                print s+'[{0:2d} :   ]    '.format(triada[0]) + ' = ',
s[triada[0]:triada[1]]
            elif triada[0] != None and triada[1] != None:
                # 11
                print s+'[{0:2d} :{1:2d} ]    '.format(triada[0],triada[1]) + ' = ',
s[triada[0]:triada[1]]

        elif len(triada) == 1:
            print s+'[{0:2d} ]        '.format(triada[0]) + ' = ', s[triada[0]]


if __name__ == '__main__':
    # Change "s" to what ever string you like, make it 9 characters for
    # better representation.
    s = 'COMPUTERS'

    # add to this list different lists to experement with indexes
    # to represent ex. s[::], use s[None, None,None], otherwise you get an error
    # for s[2:] use s[2:None]

    lista = [[4,7],[2,5,2],[-5,1,-1],[4],[-4,-6,-1], [2,-3,1],[2,-3,-1], [None,None,-1],
[-5,None],[-5,0,-1],[-5,None,-1],[-1,1,-2]]

    slicegraphical(s, lista)
```

You can run this script and experiment with it, below is some samples that I got from the script.

```
  +---+---+---+---+---+---+---+---+---+
  | C | O | M | P | U | T | E | R | S |
  +---+---+---+---+---+---+---+---+---+
  0   1   2   3   4   5   6   7   8   9
 -9  -8  -7  -6  -5  -4  -3  -2  -1

COMPUTERS[ 4 : 7 ]     =  UTE
COMPUTERS[ 2 : 5 : 2 ] =  MU
COMPUTERS[-5 : 1 :-1 ] =  UPM
COMPUTERS[ 4 ]         =  U
COMPUTERS[-4 :-6 :-1 ] =  TU
COMPUTERS[ 2 :-3 : 1 ] =  MPUT
COMPUTERS[ 2 :-3 :-1 ] =
COMPUTERS[   :   :-1 ] =  SRETUPMOC
COMPUTERS[-5 :   ]     =  UTERS
COMPUTERS[-5 : 0 :-1 ] =  UPMO
COMPUTERS[-5 :   :-1 ] =  UPMOC
COMPUTERS[-1 : 1 :-2 ] =  SEUM
[Finished in 0.9s]
```

When using a negative step, notice that the answer is shifted to the right by 1.

answered Oct 18 '14 at 17:40

mahmoh
742 ● 7 ● 14

14

My brain seems happy to accept that `lst[start:end]` contains the `start`-th item. I might even say that it is a 'natural assumption'.

But occasionally a doubt creeps in and my brain asks for reassurance that it does not contain the `end`-th element.

In these moments I rely on this simple theorem:

```
for any n,    lst = lst[:n] + lst[n:]
```

This pretty property tells me that `lst[start:end]` does not contain the `end`-th item because it is in `lst[end:]`.

Note that this theorem is true for any `n` at all. For example, you can check that

```
lst = range(10)
lst[:-42] + lst[-42:] == lst
```

returns `True`.

answered May 26 '16 at 8:16

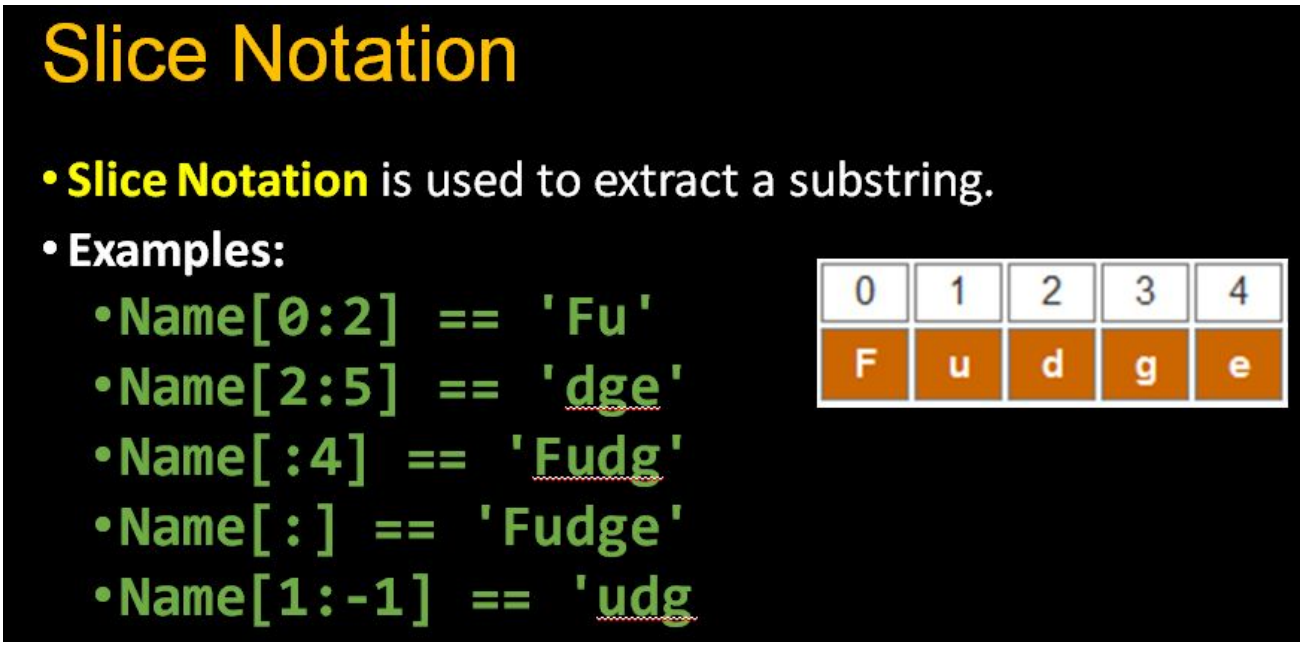
Robert

1,112 ● 11 ● 17

11

Most of the previous answers clears up questions about slice notation.

The extended indexing syntax used for slicing is `aList[start:stop:step]`, and basic examples are:



:

More slicing examples: [15 Extended Slices]

edited Sep 26 '19 at 8:19

Peter Mortensen

25.7k ● 21 ● 90 ● 118

answered Oct 6 '17 at 22:30

Roshan Bagdiya

1,256 ● 12 ● 35

11

In my opinion, you will understand and memorize better the Python string slicing notation if you look at it the following way (read on).

Let's work with the following string ...

```
azString = "abcdefghijklmnopqrstuvwxyz"
```

For those who don't know, you can create any substring from `azString` using the notation `azString[x:y]`

Coming from other programming languages, that's when the common sense gets compromised. What are x and y?

I had to sit down and run several scenarios in my quest for a memorization technique that will help me remember what x and y are and help me slice strings properly at the first attempt.

My conclusion is that x and y should be seen as the boundary indexes that are surrounding the strings that we want to extra. So we should see the expression as `azString[index1, index2]` or even more clearer as `azString[index_of_first_character, index_after_the_last_character]`.

Here is an example visualization of that ...

```
Letters   a b c d e f g h i j ...
          ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
              ¦             ¦
Indexes   0 1 2 3 4 5 6 7 8 9 ...
              ¦             ¦
cdefgh     index1        index2
```

So all you have to do is setting index1 and index2 to the values that will surround the desired substring. For instance, to get the substring "cdefgh", you can use `azString[2:8]`, because the index on the left side of "c" is 2 and the one on the right size of "h" is 8.

Remember that we are setting the boundaries. And those boundaries are the positions where you could place some brackets that will be wrapped around the substring like this ...

a b **[** c d e f g h **]** i j

That trick works all the time and is easy to memorize.

edited Jan 8 at 16:12

answered Dec 12 '17 at 4:13

asiby

1,971 ● 18 ● 24

10

In Python, the most basic form for slicing is the following:

```
l[start:end]
```

where `l` is some collection, `start` is an inclusive index, and `end` is an exclusive index.

```
In [1]: l = list(range(10))

In [2]: l[:5] # First five elements
Out[2]: [0, 1, 2, 3, 4]

In [3]: l[-5:] # Last five elements
Out[3]: [5, 6, 7, 8, 9]
```

When slicing from the start, you can omit the zero index, and when slicing to the end, you can omit the final index since it is redundant, so do not be verbose:

```
In [5]: l[:3] == l[0:3]
Out[5]: True

In [6]: l[7:] == l[7:len(l)]
Out[6]: True
```

Negative integers are useful when doing offsets relative to the end of a collection:

```
In [7]: l[:-1] # Include all elements but the last one
Out[7]: [0, 1, 2, 3, 4, 5, 6, 7, 8]

In [8]: l[-3:] # Take the last three elements
Out[8]: [7, 8, 9]
```

It is possible to provide indices that are out of bounds when slicing such as:

```
In [9]: l[:20] # 20 is out of index bounds, and l[20] will raise an IndexError exception
Out[9]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [11]: l[-20:] # -20 is out of index bounds, and l[-20] will raise an IndexError
exception
Out[11]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Keep in mind that the result of slicing a collection is a whole new collection. In addition, when using slice notation in assignments, the length of the slice assignments do not need to be the same. The values before and after the assigned slice will be kept, and the collection will shrink or grow to contain the new values:

```
In [16]: l[2:6] = list('abc') # Assigning fewer elements than the ones contained in the
sliced collection l[2:6]

In [17]: l
Out[17]: [0, 1, 'a', 'b', 'c', 6, 7, 8, 9]

In [18]: l[2:5] = list('hello') # Assigning more elements than the ones contained in the
sliced collection l [2:5]

In [19]: l
Out[19]: [0, 1, 'h', 'e', 'l', 'l', 'o', 6, 7, 8, 9]
```

If you omit the start and end index, you will make a copy of the collection:

```
In [14]: l_copy = l[:]
```

```
In [15]: l == l_copy and l is not l_copy
Out[15]: True
```

If the start and end indexes are omitted when performing an assignment operation, the entire content of the collection will be replaced with a copy of what is referenced:

```
In [20]: l[:] = list('hello...')

In [21]: l
Out[21]: ['h', 'e', 'l', 'l', 'o', '.', '.', '.']
```

Besides basic slicing, it is also possible to apply the following notation:

```
l[start:end:step]
```

where `l` is a collection, `start` is an inclusive index, `end` is an exclusive index, and `step` is a stride that can be used to take every *nth* item in `l`.

```
In [22]: l = list(range(10))

In [23]: l[::2] # Take the elements which indexes are even
Out[23]: [0, 2, 4, 6, 8]

In [24]: l[1::2] # Take the elements which indexes are odd
Out[24]: [1, 3, 5, 7, 9]
```

Using `step` provides a useful trick to reverse a collection in Python:

```
In [25]: l[::-1]
Out[25]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

It is also possible to use negative integers for `step` as the following example:

```
In[28]:  l[::-2]
Out[28]: [9, 7, 5, 3, 1]
```

However, using a negative value for `step` could become very confusing. Moreover, in order to be Pythonic, you should avoid using `start`, `end`, and `step` in a single slice. In case this is required, consider doing this in two assignments (one to slice, and the other to stride).

```
In [29]: l = l[::2] # This step is for striding

In [30]: l
Out[30]: [0, 2, 4, 6, 8]

In [31]: l = l[1:-1] # This step is for slicing

In [32]: l
Out[32]: [2, 4, 6]
```

edited Sep 26 '19 at 8:16 Peter Mortensen

answered Sep 4 '17 at 16:00 lmiguelvargasf

 **25.7k** ● 21 ● 90 ● 118  **27.7k** ● 23 ● 125 ● 137

10

I want to add one *Hello, World!* example that explains the basics of slices for the very beginners. It helped me a lot.

Let's have a list with six values `['P', 'Y', 'T', 'H', 'O', 'N']`:

```
+---+---+---+---+---+---+
| P | Y | T | H | O | N |
+---+---+---+---+---+---+
  0   1   2   3   4   5
```

Now the simplest slices of that list are its sublists. The notation is `[<index>:<index>]` and the key is to read it like this:

```
[ start cutting before this index : end cutting before this index ]
```

Now if you make a slice `[2:5]` of the list above, this will happen:

```
        |           |
+---+---|---+---+---|---+
| P | Y | T | H | O | N |
+---+---|---+---+---|---+
  0   1 | 2   3   4 | 5
```

You made a cut **before** the element with index `2` and another cut **before** the element with index `5`. So the result will be a slice between those two cuts, a list `['T', 'H', 'O']`.

edited Sep 26 '19 at 8:23
 Peter Mortensen
25.7k ● 21 ● 90 ● 118

answered Apr 4 '18 at 9:52
 Jeyekomon
1,285 ● 1 ● 15 ● 23

10

I personally think about it like a `for` loop:

```
a[start:end:step]
# for(i = start; i < end; i += step)
```

Also, note that negative values for `start` and `end` are relative to the end of the list and computed in the example above by `given_index + a.shape[0]`.

edited Jan 15 at 12:29
 Arsen Khachaturyan
4,847 ● 3 ● 27 ● 31

answered Aug 23 '19 at 14:10
 Raman
2,213 ● 1 ● 16 ● 39

8

The below is the example of an index of a string:

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1

str="Name string"
```

Slicing example: [start:end:step]

```
str[start:end] # Items start through end-1
str[start:]    # Items start through the rest of the array
str[:end]      # Items from the beginning through end-1
str[:]         # A copy of the whole array
```

Below is the example usage:

```
print str[0] = N
print str[0:2] = Na
print str[0:7] = Name st
print str[0:7:2] = Nm t
print str[0:-1:2] = Nm ti
```

edited Sep 26 '19 at 8:10

Peter Mortensen
25.7k ● 21 ● 90 ● 118

answered Jul 28 '17 at 10:12

Prince Dhadwal
205 ● 3 ● 5

5

If you feel negative indices in slicing is confusing, here's a very easy way to think about it: just replace the negative index with `len - index`. So for example, replace -3 with `len(list) - 3`.

The best way to illustrate what slicing does internally is just show it in code that implements this operation:

```
def slice(list, start = None, end = None, step = 1):
  # Take care of missing start/end parameters
  start = 0 if start is None else start
  end = len(list) if end is None else end

  # Take care of negative start/end parameters
  start = len(list) + start if start < 0 else start
  end = len(list) + end if end < 0 else end

  # Now just execute a for-loop with start, end and step
  return [list[i] for i in range(start, end, step)]
```

edited Sep 26 '19 at 8:22

Peter Mortensen
25.7k ● 21 ● 90 ● 118

answered Dec 19 '17 at 6:12

Shital Shah
37.4k ● 7 ● 160 ● 123

The basic slicing technique is to define the starting point, the stopping point, and the step size - also known as stride.

4

First, we will create a list of values to use in our slicing.

Create two lists to slice. The first is a numeric list from 1 to 9 (List A). The second is also a numeric list, from 0 to 9 (List B):

```
A = list(range(1, 10, 1)) # Start, stop, and step
B = list(range(9))

print("This is List A:", A)
print("This is List B:", B)
```

Index the number 3 from A and the number 6 from B.

```
print(A[2])
print(B[6])
```

Basic Slicing

Extended indexing syntax used for slicing is aList[start:stop:step]. The start argument and the step argument both default to none - the only required argument is stop. Did you notice this is similar to how range was used to define lists A and B? This is because the slice object represents the set of indices specified by range(start, stop, step). Python 3.4 documentation.

As you can see, defining only stop returns one element. Since the start defaults to none, this translates into retrieving only one element.

It is important to note, the first element is index 0, *not* index 1. This is why we are using 2 lists for this exercise. List A's elements are numbered according to the ordinal position (the first element is 1, the second element is 2, etc.) while List B's elements are the numbers that would be used to index them ([0] for the first element 0, etc.).

With extended indexing syntax, we retrieve a range of values. For example, all values are retrieved with a colon.

```
A[:]
```

To retrieve a subset of elements, the start and stop positions need to be defined.

Given the pattern aList[start:stop], retrieve the first two elements from List A.

edited Sep 26 '19 at 8:25

Peter Mortensen
25.7k ● 21 ● 90 ● 118

answered Jul 23 '18 at 13:06

Babu Chandermani
81 ● 1 ● 5

3

I don't think that the Python tutorial diagram (cited in various other answers) is good as this suggestion works for positive stride, but does not for a negative stride.

This is the diagram:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
```

```
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

From the diagram, I expect `a[-4,-6,-1]` to be `yP` but it is `ty`.

```
>>> a = "Python"
>>> a[2:4:1] # as expected
'th'
>>> a[-4:-6:-1] # off by 1
'ty'
```

What always work is to think in characters or slots and use indexing as a half-open interval -- right-open if positive stride, left-open if negative stride.

This way, I can think of `a[-4:-6:-1]` as `a(-6,-4]` in interval terminology.

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
  0   1   2   3   4   5
 -6  -5  -4  -3  -2  -1

+---+---+---+---+---+---+---+---+---+---+---+---+
| P | y | t | h | o | n | P | y | t | h | o | n |
+---+---+---+---+---+---+---+---+---+---+---+---+
 -6  -5  -4  -3  -2  -1   0   1   2   3   4   5
```

answered May 27 '19 at 20:25

aguadopd
472 ● 5 ● 13

Highly active question. Earn 10 reputation in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.